A Process for Verifying and Validating Requirements for Fault Tolerant Systems using Model Checking[1]

Francis Schneider[2], Steve M. Easterbrook, John R. Callahan and Gerard J. Holzmann[3]
William K. Reinholtz[2], Adins, Ko[2], Mohammad Shahabuddin[2]
NASA/WVU Software Research Lab
100 University Drive, Fairmont, West Virginia 26505

Abstract

Model checking is shown to be an effective tool in validating the behavior of a fault tolerant embedded spacecraft controller. The case study presented here shows that by judiciously abstracting away extraneous complexity, the state space of the model could be exhaustively searched allowing critical functional requirements to be validated down to the design level. Abstracting away detail not germane to the problem of interest leaves by definition a partial specification behind. The success of this procedure shows that it is feasible to effectively validate a partial specification with this technique. Three anomalies were found in the system. One was an error in the detailed requirements, and the other two were missing/ambiguous requirements. Because the method allows validation of partial specifications, it is also an effective approach for maintaining fidelity between a co-evolving specification and an implementation. We also show that two of the three anomalies were found in the implementation, demonstrating the overall effectiveness of the process and the importance of a good software design.

[1] *The research described in this paper was carried out in part by the Jet Propulsion Laboratory, California Institute of Technology, under a contract with the National Aeronautics and Space Administration, and in part by West Virginia University under NASA cooperative agreement #NCC 2-979. Reference herein to any specific commercial product, process, or service by trade, name, trademark, manufacturer, or otherwise, does not constitute or imply its endorsement by the United States Government, the Jet Propulsion Laboratory, California Institute of Technology or West Virginia University*

[2] Jet Propulsion Laboratory / California Institute of Technology,
MS 125-233 Pasadena, CA 91109.

[3] Computing Sciences Research, Bell Laboratories, Lucent Technologies, Murray Hill, NJ 07974

1 Introduction

This paper describes a practical application of model checking for validating the requirements for a complex embedded system. The case study described here is of a dually redundant spacecraft controller, in which a checkpoint and rollback scheme is used to provide fault tolerance during the execution of critical control sequences. The challenge given to us and the purpose of this study was to determine if model checking could be used to uncover errors in an existing design specification for a space craft system. To this end the project manager supplied us with such a specification.

The software requirements specification for the spacecraft specifies the required behavior for the checkpoint and rollback scheme. However, the validity of these requirements could not be determined through inspection. In other words, it was not possible to determine whether the behavior described in these requirements would provide the desired level of fault tolerance. More importantly, testing of the eventual implementation would not necessarily provide this validation either, due to the difficulty of ensuring test case coverage for all possible fault occurrence scenarios.

The approach described here uses a formal automata-based model derived from the specification. We used various high-level safety properties to validate the generalized system model. Key system functional requirements were then validated by defining corresponding liveness properties in linear temporal logic, which were required to be satisfied when the system responds to errors. We used the model checker Spin [2] to identify traces in the model for which these properties were violated.

The work described in this paper forms part of an on-going investigation into lightweight formal methods for V&V of requirements specifications. We use the term 'lightweight' to indicate that the methods can be used to perform partial analysis on partial specifications, without a commitment to developing and baselining complete, consistent formal specifications. The formal methods are used to model critical chunks of an informal specification, to check that key properties hold. The aim is to find errors, rather than to prove correctness. Application of the methods is driven by the needs of the project, and is used as a modeling tool to answer questions that arise during verification and validation.

The paper is organized as follows. Section 2 provides a motivation for the case study by briefly surveying existing approaches to requirements validation and demonstrating why these approaches do not provide the desired level of assurance. We introduce the distinction between verifying requirements through completeness and consistency checking, and validating requirements against real world properties ('claims') that should follow if the statement of the requirements is correct.

Section 3 introduces the dually redundant system, and shows how it was expressed as a FSM. We show how the system behaves as a communications system, making it particularly amenable to analysis using the model checker Spin.

Section 4 describes the steps that were taken to optimize the model, in order to reduce the size of the state space. We show how the model was partitioned into five separate fault scenarios, and explain in detail how one of these scenarios was checked. We discuss the process of checking the model against claims expressed as linear temporal logic formulae. Section 5 presents the results of the analysis.

Section 6 provides a discussion of the results, including a reflection on the benefits seen in the case study. The importance of partitioning the model in order to make the analysis feasible is discussed, along with some reflection on the resulting limitation of the analysis ('partial analysis of partial specifications').

Section 7 presents conclusions and describes our future work. A short overview of the theoretical basis for the use of the LTL and Büchi automata is provided in appendix A.

2 Background

Requirements validation is the process of determining that the specified requirements capture the real world needs of the stakeholders. For real-time control systems, this involves checking that the specified behavior will in fact provide safe and effective control, without introducing any undesirable effects. For reasonably complex systems, validity of the requirements is hard to establish. Informal methods only provide a very basic level of assurance, by imposing a structure on the specification that facilitates inspection by domain experts. Formal methods have the potential to provide a much greater level of assurance, through the construction of a precise model of the requirements, which can be tested against domain properties.

A number of formal modeling tools are available that are applicable to software systems. Heitmeyer and Mandrioli [3] provide an excellent overview of the current state of the art. Here we concentrate on state machine models, which can be used to test safety and liveness properties.

RSML [1, 3] and SCR [5] have both been very successful at providing static analysis techniques for checking completeness and consistency of specifications expressed as deterministic state machines. However, fault tolerant systems are inherently non-deterministic, that is, the transition schemes are relational not functional. Systems with inherent non-determinism are not easily amenable to analytic static evaluation methods. Systems that can be partitioned into a deterministic and a non-deterministic part can apply tools such as RSML or SCR to validate deterministic components. For example, Easterbrook [6] has reported using the SCR tool in this way to validate the Fault Detection, Isolation and Recovery (FDIR) requirements for a spacecraft bus controller. The deterministic part was modeled in SCR, and then extended to include non-deterministic elements (i.e. fault occurrences) using the Spin model checker [2]. Such a procedure would be suggested for example when an otherwise deterministic system had to be shown to be resilient under (non-deterministic) fault injection.

An analysis based methodology such as RSML or SCR requires determinism in the underlying model to prove requirements completeness and consistency. In contrast, state space exploration methods ('model checking') are operational in nature rather than analytic. They allow functional requirements to be validated over non-deterministic finite state machines using optimized reachability schemes. By incorporating functional requirements in a non-deterministic model, requirements properties can be validated. Manna and Pneuli [6] have shown that virtually any expressible requirements property can be represented as a safety, precedence, or liveness property using the Linear Temporal Logic (see appendix A).

Three such model checkers have been widely used for verification of low-level designs of both hardware and software, and communication protocols. The Murphi model checker has a rich support for temporal logic and allows invariants to be expressed in the model to be checked as the state space exploration evolves. It supports a single site model only, which is a disadvantage in the validation of concurrent systems. The Symbolic Model Verifier (SMV) has been applied successfully to communication protocols [8]. SMV can validate synchronous and asynchronous systems against a system specification specified in the temporal logic CTL [10] [2]. It allows for non-determinism in the specifications and for concurrency in the model within procedures. It supports rich temporal logic specifications but does not support complex data structures, making it difficult to build a complete low level model. Both SMV and Murphi were designed for validating hardware systems. The Spin model checker was designed for verification of communication protocols, and provides support for a basic set of software data structures.

Tool	Deterministic?	Counter Example Generation?	Reqts. expressible as LTL Formulae?	Developed for V&V of:
RSML	Y	N	N	TCAS S/W
SCR	Y	Y	N	A7e Aircraft S/W
Murphi	N	Y	Y	Single Process S/W
SMV	N	Y	Y	Comms H/W
Spin	N	Y	Y	Comms S/W

Table 1: System Validation Tools

Each of the three model checkers permits a rich set of temporal logic formulae to be incorporated into the modeling system. We chose to use the Spin model checking system for this study because it (a) was designed to validate software communications protocols (a) is algorithmic in nature (c) supports data structures allowing detail where appropriate (d) incorporates linear temporal logic primitives allowing functional requirements to be validated over the model (e) and, significantly, because the modeling system can be used to validate functional requirements over traces from the implementation.

3 DRS High Level Model Description

The case study described here is a Dually Redundant System (DRS) for a spacecraft controller, consisting of two hardware platforms running identical software to maximize system reliability and availability. The systems exchange information to synchronize software operation. One of the systems has control of the system bus and is called the *prime string*. The other, known as the *online string*, provides a backup, executing in synchronization or at most within one second of the prime string. Information is exchanged between the two systems by the synchronous (rendezvous) communication of a 32-word table, the State Table Broadcast (STB), broadcast by the prime string once per second. The online string uses this to keep itself in synchronization with the prime string.

The system executes high priority programs called *critical sequences* that must be tolerant of arbitrary faults. To this end, the strings use a variant of the checkpoint and rollback process found to work well in industrial applications [11]. Checkpoints correspond to completed transactions in the executable code. Such a completion is referred to as a commit operation, meaning that if a system crash occurs, system operations could be rolled back to the point where the commit occurred and proceed from here. The spacecraft controller works analogously except that the checkpoints are referred to as markpoints, and are hard coded into the executing program.

For example, consider the retrieval and return of a soil sample by a remote robot. Successful retrieval of the sample is an operation that need not be repeated. The code ending in the completion of this process would be delineated with a markpoint. The next group of instructions might be the storage of the sample that was just retrieved, at the end of which would be another markpoint. If any operation were interrupted by the occurrence of a fault, the system would repair the fault; roll back control to the beginning of the last markpoint; and continue execution from there. It would not be necessary to waste battery power or time to retrieve another sample if that was already achieved. This paper focuses on the validation of the fault tolerance provided by this mark and rollback process.

The fault containment requirements specify that fault protection shall operate only in the prime string. While the prime string is repairing a fault, the online string must stop executing its copy of the critical sequence and wait for the STB to tell it that the fault has been repaired, thereby signaling it to proceed with the critical sequence.

The rollback requirements specify that three full seconds of execution time shall be allowed to pass after a new markpoint is encountered by the software before the new markpoint is recognized as a legitimate rollback point. This is because the system controls external elements that are mostly mechanical in nature. Accordingly, the software is, in general, always ahead of the hardware. The three-second delay gives any mechanical tasks a chance to be completed, and for any faults that occurred to be properly logged, before the previous section of the critical sequence can be considered successfully complete. To implement this requirement, each new markpoint is aged each second by one second by moving it one level deeper in a three-level buffer. Only markpoints that have reached the bottom will be eligible for use in the rollback process. Figure 1 shows a high level snapshot of normal critical sequence operation in both strings.

4 Validation Procedure

4.1 Modeling

The first step was to produce a state model of the DRS system. To model the specified behavior, we treated the mark and rollback process as a communications system. Holzmann [12] has defined a communications protocol as a five component specification for how communication is to be carried out in an error free way among two or more separate elements. For the mark and rollback process, these properties are:

1. *The service provided* by the protocol is to keep the prime and the online systems in synchronization. This is done so that the online string can take over quickly should the prime system become inoperable.
2. *The environmental assumptions* are that the prime string interacts with an entity that provides information about faults.
3. *The major vocabulary* consists of the variables SFP, CS, and CM. SFP is the Spacecraft Fault Protection flag. When this flag is set, the system has experienced a fault that has not yet been repaired. The CS flag is set in the prime string and in the backup string when the critical sequence is active i.e. running in each respective string. The CM flag is set to indicate that the critical sequence is active or in standby pending the repair of a fault and accordingly to remind the strings that when an interfering fault is fixed, the suspended critical sequence needs to be restarted at the last valid aged markpoint.
4. The three protocol flags each use single bit encoding, as shown in Table 2.
5. *The procedure rules* are most complex to deal with, the hardest to specify, the most difficult to validate. Most of the validation work occurs here. Examples from the mark and rollback support application are that the protocol variables SFP, CM, and CS are to be broadcast once each second to the online string and actually also back to the prime string by the prime string to allow the prime string to check its own synchronization.

Flag	Value	Meaning
SFP	1	Fault
	0	Cleared
CS	1	CS executing
	0	CS not executing
CM	1	CS active or suspended
	0	CS inactive and not suspended

Table 2: Communication Flags

The initial model was represented using statecharts [13]. Figures 1 and 2 show portions of the statecharts for the prime and online strings respectively.

In the case study presented here, certain types of faults are of such a nature that they can be repaired by the prime string. When a fault occurs, the three protocol flags (CS, CM, SFP) change state from (1, 1, 0) to (0, 1, 1). This information is broadcast to the online string once per second. When the online string sees the SFP flag is set, it suspends operation of the executing critical sequence and waits for the prime string to repair the fault. Once the fault is repaired, the prime string can roll back to the last valid markpoint and resume processing. The online string will see the new SFP flag is reset in the STB message, rollback to the aged broadcast markpoint and restart its copy of the critical sequence.

This example shows a small subset of the actual elements and their procedure rules that belong in each category. The complete protocol specification is in excess of 80 pages.

4.2 Estimation of State Space Size

Once an initial model is obtained, the state space size must be estimated, in order to assess the potential for automated validation. This was done by estimating the number of substates needed in the Spin model to implement each state shown on the statecharts. For example, the full statechart for the prime string has 16 states and each could be implemented with, say, 4 substates giving a state space of $4 \times 4 \times \ldots \times 4 = 2^{32}$.

The full statechart for the online string has 14 states. Assuming 4 substates for each gives $4 \times 4 \times \ldots \times 4 = 2^{28}$ states. The rendezvous communication contains 32 data elements, 5 of which are unused leaving a total of 27 elements. Each of these remaining 27 is at least a binary flag. This gives $2 \times 2 \times 2 \ldots \times 2 = 2^{27}$ states as a minimum. This contributes to the state space of the online string, giving $2^{28} \times 2^{27} = 2^{55}$ states total.

Both strings operating as one system will have a state space of:

$$(2^{32} \text{ states prime string}) \times (2^{55} \text{ states online string}) = 2^{87} \text{ states.}$$

With a CPU that executes 1 state per microsecond, the system will traverse its reachability graph in about 10^{12} years.

The problem of interest here is to discover the failure modes of this system. To be able to do this we must reduce the state space down to an manageable size by abstracting away states that are not germane to the operation we are interested in, namely (a) the repair of faults (b) the rollback process and (c) the synchronization between the prime and the online (backup) systems. The result is a

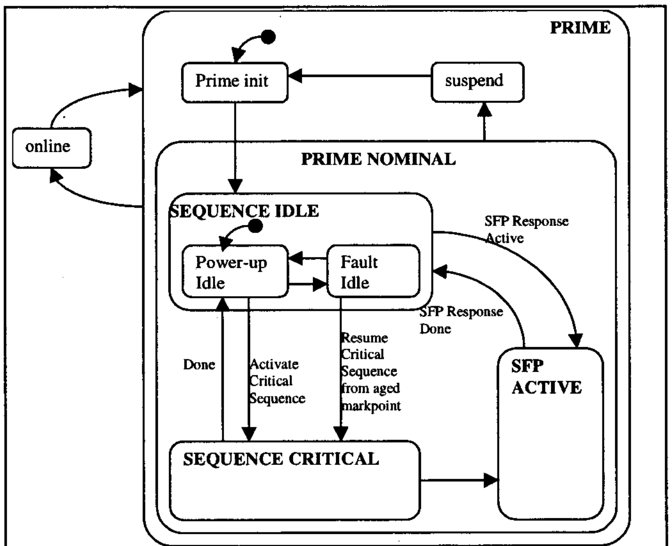

Figure 1: A partial statechart for the DRS prime string

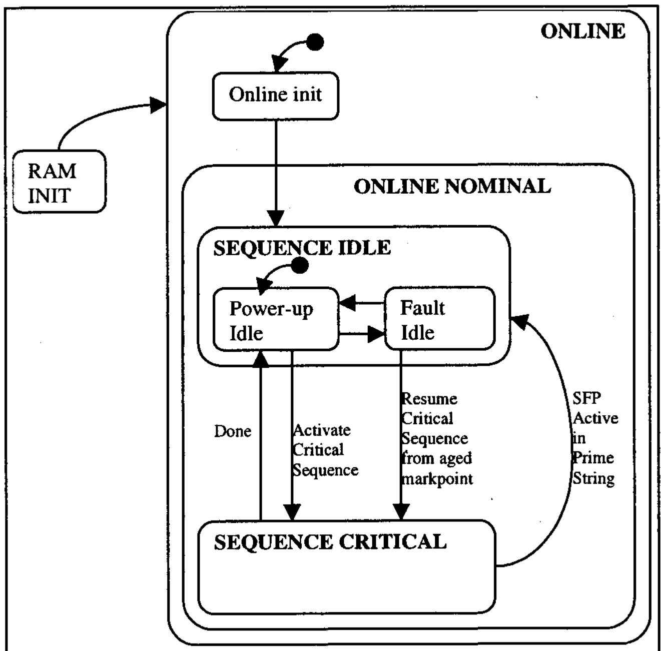
ONLINE
Online init
RAM INIT
ONLINE NOMINAL
SEQUENCE IDLE
Power-up Idle
Fault Idle
Done
Activate Critical Sequence
Resume Critical Sequence from aged markpoint
SFP Active in Prime String
SEQUENCE CRITICAL

Figure 2: A partial statechart for the DRS online string

partial specification, but which has enough detail left to partially validate the properties of interest.

4.3 Reducing the state space

There are a number of ways in which the state space can be reduced to a size amenable to model checking. Firstly, the functional requirements of the system may be partitioned into equivalence classes, by exploiting natural symmetries or subclasses that may be present in the domain. Secondly, the validation task can be partitioned by separately validating requirements that are known to be independent from one another. Validation of each requirement in isolation should traverse less of the overall state space than all of the requirements taken together. In either case, detail that is not germane to each validation task can be temporarily removed from the model. We will illustrate each of these approaches below.

For the DRS system, we partitioned the functional behavior by separating out the classes of fault that can occur. The requirements include a simplifying assumption that facilitated this partitioning:

Fault protection shall be designed assuming only one fault occurs at a time, and that a subsequent fault will occur no earlier than the response completion time for the first fault, and that multiple detections occurring within the response time are symptoms of the original fault.

The requirements identify 5 classes of faults that can occur on the spacecraft. Accordingly, the Mark and Rollback process can be partitioned into five equivalence classes. Each can be treated independently of the others, significantly reducing the size of the overall state space to be checked by the validation process. We also exploited the symmetry between the redundant processors running the online and the prime strings, by recognizing that either string could run on either processor.

The five fault classes are as follows:

1. SFP Execution Non-UV Trip
2. Online Fault
3. Peripheral Interfering Fault
4. Prime Fault
5. SFP Under-Voltage Trip

In the first three cases, the Prime String will handle the fault, while both strings suspend execution of the critical sequence. In case 4, the fault is in the prime string, and the online string will take over. The online string then becomes prime. In the final case, the fault could be anywhere, so either processor may end up as prime. In all cases, once the fault protection response is complete, the critical sequence should be resumed from the last aged markpoint, by whichever processor is now prime.

Equivalence class 1 contains the fundamental mark and rollback scenario common to the other classes during normal operation and it has less structure, in that it executes the smallest subset of states in the 5 partitions considered above. We therefore used this as the first validation exercise. We will concentrate only on this class for the remainder of the paper. It will be seen that validation of this class has implications for the other requirements classes as well. We proceed first by removing all states in the statecharts that do not contribute to the mark and rollback process. The resulting states are, in fact, those shown in figures 1 and 2.

The prime string now contains 7 states and the online string 5 states. If we assume once again that as a minimum again each state can be implemented with 4 substates, then these two elements contribute

$$4^7 \times 4^5 = 16{,}777{,}216 \text{ states.}$$

The overall state space can be further reduced by ignoring the CM and CS flags. By abstracting these two flags away we will be checking only the fundamental mark and rollback process that depends upon the SFP flag and the relative position of the markpoint with respect to critical sequence execution time. If we want to learn about any possible effects of the CS and the CM flags they will have to be inserted back into the model at some point. If the state space becomes too large, a non-exhaustive search option would then have to be used.

A further strategy for reducing the state space is to reduce the complexity of the input data. The model can be validated on the simplest possible test runs, and then if no errors are uncovered, the size of the dataset can be increased gradually. In this case, the length of the critical sequence can be considered input data. A minimal critical sequence would contain the smallest number of markpoints possible. A critical sequence containing 3 markpoints was chosen for the initial exercise, as it contained sufficient complexity to determine all possible combinations of fault occurrence and rollback.

Finally, by removing the states that are not executed in fault class 1, the state space was reduced to an estimated:

$$(4\text{ prime})^4 \times (3 \times 2\text{ rendezvous packet}) \times (4\text{ online})^3 = 98{,}304\text{ states}$$

Now adding an extra flag for the presence of a fault doubles this to 196,608 states. This is still a manageable state space for the Spin tool.

4.4 Validation of Case 1

A SFP Execution Non-UV Trip is a spacecraft fault that is outside of the DRS system *per se*. These correspond to the type covered by partition 1 in this case study. In this case the prime string is given the task of repairing the fault. The prime string would set the SFP flag to 1 to indicate a fault operation is in progress; stop the running critical sequence; and enter the SFP Active state to repair the fault (see Figure 1). The STB would still be transmitted to the online string once per second. That is, since the fault is outside of the prime string, its ability to function has not been impaired. Having received the STB, the online string will cease running its copy of the critical sequence and transition to the Fault Idle state, waiting there until it receives an STB message indicating that the fault has been cleared. Once the prime string has repaired the fault it sets its SFP flag to zero and enters the Fault Idle state in preparation for resuming the critical sequence. At this point it rolls back to the last valid (aged) markpoint; and resumes executing its copy of the critical sequence at this location. When the online string sees an STB message indicating that the SFP flag is 0, it enters the SEQUENCE CRITICAL state resuming execution of its copy of the critical sequence at the aged broadcast markpoint.

The first step in the validation is to develop Linear Temporal Formulae representing the requirements to be validated. Each LTL formula is then incorporated into the resulting Spin model as a "never" clause. Details of the validation method are described in Appendix A.

To check that the desired fault tolerance is achieved, three separate functional requirements need to be validated in each string:

R1. If a fault occurs when the last markpoint was at the start of the program, the prime string shall roll back to the start regardless of how much time has expired since the program started running.

R2. If a fault occurs when the time t following the last markpoint was less than 3 seconds and the last markpoint was not at the start of the program, the prime string shall roll back to the next previous markpoint. That is, do not use the markpoint that has not yet been properly aged, even though it has been encountered in the execution of the current critical sequence.

R3. If a fault occurs when the time t following the last markpoint was greater than or equal to 3 seconds the prime string shall roll back to the last valid aged markpoint.

Requirements R4, R5, and R6 express the same three requirements for the online string. These can all be expressed as liveness conditions; they specify an action that must take place now or in the future. Symbolically, the LTL formulae representing these conditions have the form:

$$\Diamond\, p \wedge \Box(p \rightarrow \Diamond\, q)$$

Where p is the occurrence of a fault, and q is the correct response. $\Diamond$ and $\Box$ are the temporal logic operators ‘eventually’ and ‘always’. $\Diamond x$ means at some future state x will be true. $\Box x$ means the x is true in the current state and in all future states. The formula expresses the condition that eventually a fault (p) does occur, and that it is always true that when it occurs, at some point in the future the correct rollback operation (q) will occur.

Note that strictly speaking, our validation of the requirements only involves the latter part of this formula, i.e. $\Box(p \rightarrow \Diamond\, q)$, as we are checking that the correct rollback eventually occurs in response to a fault. However, this is trivially true if no faults ever occur (i.e. if p is never true). Hence we add the condition $\Diamond p$ to check that our fault injection model does indeed inject this type of fault. This removes the possibility of false positives during the validation exercise.

The LTL equivalent of requirement R1 is as follows:

$$\Diamond\, p \wedge \Box(p \rightarrow \Diamond\, q) \qquad \text{(R1)}$$

where $p = (\text{SFP} = 1) \wedge (\text{markpoint} = \text{start})$

and $q = (pc = \text{markpoint}) \wedge (SFP = 0)$

Where markpoint is the default markpoint address of the beginning of the sequence; pc is the critical sequence machine program counter; and start is the address of the beginning of the critical sequence program. Note that requirement R1 said nothing about *how quickly* the rollback should occur, and neither does our formalization, as we have not said anything about the intervening states between the fault, p, and the rollback, q. We could define further temporal formulae to investigate such concerns at a later stage in the analysis.

Requirement R2 becomes:

$$\Diamond r \wedge \Box(r \rightarrow \Diamond s) \qquad \text{(R2)}$$

where $r = (t < 3) \wedge (SFP = 1) \wedge (\text{mp\_current} \neq \text{start})$

and $s = (pc = \text{mp\_next\_previous}) \wedge (SFP = 0)$

and R3 becomes:

$$\Diamond u \wedge \Box(u \rightarrow \Diamond v) \qquad \text{(R3)}$$

where $u= (t \geq 3) \wedge (SFP=1) \wedge (\text{mp\_current}=\text{mp\_ge\_three\_sec})$

and $v = (pc = \text{mp\_current}) \wedge (SFP = 0)$

Where t is time in seconds since the last encountered markpoint; here mp_current represents the current markpoint and mp_previous represents the markpoint preceding mp_current; each of these represents the case where less than three seconds have expired. mp_ge_three_sec represents the markpoint for the case where three or more seconds have expired since the last encounter of a markpoint in the sequence.

Three analogous requirements are needed for the online string, using its copies of SFP and Mark:

$$\Diamond h \wedge \Box(h \rightarrow \Diamond i) \qquad \text{(R4)}$$

$$\Diamond j \wedge \Box(j \rightarrow \Diamond k) \qquad \text{(R5)}$$

$$\Diamond l \wedge \Box(l \rightarrow \Diamond m) \qquad \text{(R6)}$$

Each additional LTL formula that is added to the model adds more complexity, making runtimes and memory consumption very large. The best way to circumvent this problem is to validate each functional requirement separately. For example, we can check that requirement R1 is satisfied without looking at R2 and R3 because they are independent requirements. However, requirement R1 is not independent of R4. This non-orthogonality requires that both be validated in the same run. Semantically, this means that when rollback takes place in the prime string under the condition that we are at the start of the program, then the same rollback must be also shown to take place in the online string. The derivation in Appendix A shows that a jointly operational Büchi Automaton can be produced from separate LTL formula by writing down the logical conjunction of the formulae and then converting the result to an equivalent automaton. The conversion itself is done with the Spin option -f and is automatic, although the user may want to apply a certain amount of optimization on the result to make the resulting automaton more efficient. To keep the resulting system at a minimum, the automaton for rollback to the beginning of the program is derived from R1 and R4:

$$\Diamond p \wedge \Box(p \rightarrow \Diamond q) \wedge \Diamond h \wedge \Box(h \rightarrow \Diamond i) \qquad \text{(R7)}$$

where p, q, h and i are as defined above.

Analogous minimal LTL formulae were derived for the other 3 cases and they were implemented in the model.

Additional validation can be performed by defining further properties that should hold in the model. For example, we could check that aged markpoints are always in agreement with each other. This condition can be stated by using the safety condition that the aged markpoint x in the prime string never disagree with the aged broadcast markpoint y in the online string. The corresponding safety condition would be

$$\Box(x = y) \qquad \text{(R8)}$$

Additionally, assertions were used throughout the model to confirm that the model had the desired behavior.

5 Results

Five different fault categories were identified to test the model. The results reported here cover the first of these categories only (partition 1), but we do discuss implications for the other five fault categories. Fault category 1 refers to the behavior of the DRS prime string in the face of a SFP Execution Non-UV Trip.

Six separate requirements on the rollback scheme were validated, as described in section 4.4. Each of the six requirements involved exhaustive examination of approximately 100,000 states in the model, and took about 30 seconds each. The response and recovery in each case was to the injection of a non-UV trip fault in all possible ways, based on the model. Three of the 6 runs for the 6 requirements failed in the verification.

We define a fault as an element in a system that that doesn't perform to specifications. An anomaly is incorrect information appearing in the system that is the manifestation of the fault. Three anomalies were identified and are described below. The first two are errors in the requirements that we thought might not occur in the DRS implementation. The third was a discrepancy in the detailed requirements that could allow for erroneous behavior of the implemented system.

1. Depending on how error detection and repair is handled, it may be possible for the prime system to detect and to repair an intermittent error within one second, and then consequently not broadcast this state to the online system. This would mean that the online system would not receive notice of the fault; therefore, it would continue executing its copy of the critical sequence. Repeated occurrence of this scenario would cause the online system to get ahead of the prime system, possibly to the point where the online system would complete its copy of the sequence. If the prime system subsequently fails, the online system may not have a markpoint to roll back to. Details are as follows. The synchronization between the prime and online systems is at one-second boundaries. The STB handshake in interrupt five will not report the occurrence of a fault that occurred after the beginning of the one-second time frame. However, if the fault is repaired before the start of the next one-second time frame, the prime string will not report it in the STB at the next handshake either. The result is that the online system is now one second ahead of the prime system. Figure 3 shows one sequence of events in this scenario. Inspection of the figure shows that repeated injection of this type of intermittent fault effectively stalls the prime execution while the online system will eventually complete executing its copy of the sequence. This anomaly is due entirely to the ordering of processing described in the design. Figure 3 shows a detailed example of how this condition could develop after the detection and repair of a single fault in the prime string. The three aging buffers in the prime string and the two buffers in the online string show a new mark point (location 6) having been aged by one second when the fast-fault-repair scenario occurs in the prime system. The correct rollback point is that given for the prime string since it controls the space craft bus. It is shown as location 1 at the bottom of the prime aging buffer. The anomaly or resulting incorrect information in the system is the mark point identified as location 6 at the bottom of the online aging buffer.

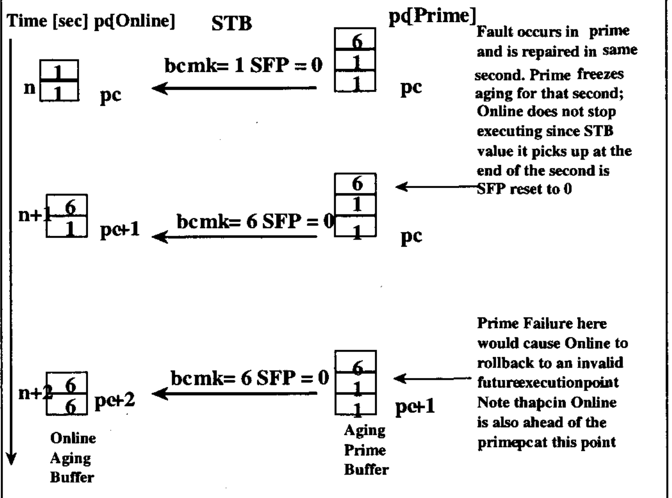
Time [sec]
pc[Online]
STB
pc[Prime]
Fault occurs in prime and is repaired in same second. Prime freezes aging for that second; Online does not stop executing since STB value it picks up at the end of the second is SFP reset to 0
n
n+1
n+2
bcmk= 1 SFP = 0
bcmk= 6 SFP = 0
bcmk= 6 SFP = 0
pc
pc+1
pc+2
pc
pc
pc+1
Prime Failure here would cause Online to rollback to an invalid futureexecutionpoint Note thatpc in Online is also ahead of the primepc at this point
Online Aging Buffer
Aging Prime Buffer

Figure 3: Anomaly One

2. The second anomaly depends upon how faults are handled at the end of a critical sequence. If a fault occurs in the prime system less than three seconds *after* the end of the critical sequence is reached, it is not clear how the rollback, if any, would be handled. The requirements specification did not designate the critical sequence end instruction as a markpoint. Our validation run failed because our model assumed that once the critical sequence completed, the online system returned to the Power Up Idle state; accordingly there would be no suspended critical sequence to return to once the fault was corrected. If the fault were to bring the prime system down, the

online system may need to roll back to the last aged markpoint. This anomaly is due to a missing requirement. Figure 4 shows a detailed example of how this condition could evolve. Both the aging buffers would have had time to develop correct rollback points shown as location 9 in both cases. However, the time sequence following the dashed line indicates that each critical sequence would have finished execution, therefore neither critical sequence would be available for follow-on rollback after fault repair. In each case both critical sequences terminated and transitioned out of their execution state. In this instance the anomaly or incorrect information in the system is loss of the critical sequence itself and its associated program counters and aging buffers.

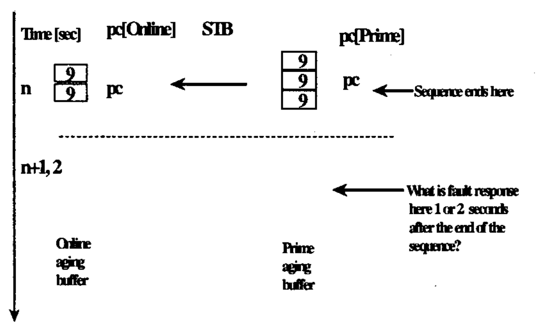

Figure 4: Anomaly Two

3. The third anomaly concerns the occurrence of a fault 2 seconds after a markpoint is encountered in the prime string. The prime system freezes the aging function at n+2 seconds. Since faults that occurred in the previous second are not broadcast to the online system until the current second, the online system will continue to execute, aging its markpoint by one further second. At this point the online system receives the SFP = 1 value and now both agers are frozen. Once the fault is repaired, the both strings will roll back, but the online system will roll back to the newer markpoint. This would not cause a problem if the prime system then completes the critical sequence. However, if the online system should subsequently have to take over due to a prime failure - possibly associated with the (symptomatic) fault that was just processed, it could roll to an inappropriate block of code. This problem would not go away if the aging buffers were made deeper or shallower. It would just occur at a different place since it is a consequence of the relative time difference between the two aging schemes. Figure 4 shows a detailed example of how this condition could develop. It shows a fault occurrence that freezes the prime ager with mark point 6 at the 2-second point. Subsequently the corresponding online string ager ages its 6 by one extra second before receiving the STB message informing it of the occurrence of the prime fault. Accordingly, the anomaly or incorrect information in the system resulting from this fault is that the 6 has made it to the bottom of the online string aging buffer yielding an invalid rollback point. The correct rollback point is to location 1, the start of the critical sequence.

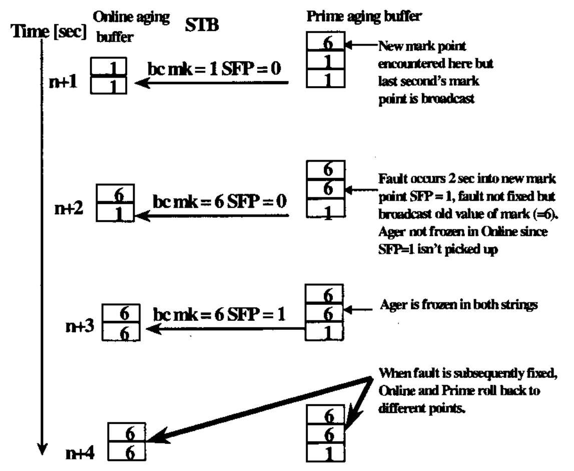
Time [sec]
Online aging buffer
STB
Prime aging buffer
n+1
1
1
bc mk = 1 SFP = 0
6
1
1
New mark point encountered here but last second's mark point is broadcast
n+2
6
1
bc mk = 6 SFP = 0
6
6
1
Fault occurs 2 sec into new mark point SFP = 1, fault not fixed but broadcast old value of mark (=6). Ager not frozen in Online since SFP=1 isn't picked up
n+3
6
6
bc mk = 6 SFP = 1
6
6
1
Ager is frozen in both strings
When fault is subsequently fixed, Online and Prime roll back to different points.
n+4
6
6
6
6
1

Figure 5: Anomaly Three

6 Validation of the Implementation

We have subsequently validated the implementation for the presence of the three design anomalies.

For this purpose we used a special purpose spacecraft simulator called the High Speed Simulator (HSS) [16,17]. The simulator uses code identical to the real spacecraft. However, it is decoupled from hardware and telemetry. Accordingly, its use as a test vehicle (1) is an accurate measure of system functionality and (2) it allows rapid turnaround on test suite creation, execution, and reporting of results.

The simulator allows test engineers to write test sequences for execution on the simulator. Given the data structures present in the spacecraft controller, a Tool command language (Tcl) program is written that orchestrates (1) the execution of the test sequence, (2) the extraction and printing of values of selected data attributes (3) the extraction and printing of any relevant time stamps and (4) fault injection scenarios and their responces.

6.1 Validation of the Implementation

The context here uses the term "prime system fault" to mean the prime system is responding to and repairing a fault that occurred in another part of the spacecraft. Additionally, the prime system must repair the fault before the sequence can be continued.

6.2 Procedural Steps

We wanted to know if the software implementation contained the same anomalies as were found in the design. To determine this, we supplied the High Speed Simulator with a simple sequence program for execution. By injecting faults into the running sequence, the same problematic conditions would be set up in the implementation that were discovered by design validation. Our earlier validation work derived the design anomalies from a three-step process. First, the prime system would stop running freezing its mark point ager in response to a fault occurrence somewhere in the spacecraft. Second, the prime system would load and begin execution of a fault recovery program. Finally, during its execution of the fault recovery program, the prime system itself would fail. To affect this same scenario in the software implementation, the prime system was commanded to do a cold boot at execution points in the implementation identical to those that caused the anomalies in the design validation. An operational online system considers the prime system cold boot to be a prime system failure. It reacts by becoming prime itself; taking control of the spacecraft bus; rolling back to the relevant earlier mark point address if necessary; and resuming execution of the sequence program. For example, the third anomaly found in the design validation process occurs when the prime system fails after encountering a fault scenario that freezes its mark point at second two in the aging process. This results in the new prime system rolling back to an inappropriate address due to a timing problem in the design. Accordingly, cold booting the prime system when it has aged its mark point by two seconds has the same effect as the two step process considered in the design case.

Detection of the presence of design anomalies in the implementation was done by selecting data structures for output identical to those used in the design case. These output data values taken together at any execution cycle represent the state of the implementation at a particular point in time. As the implementation executes, this 'state vector' describes a finite state machine that represents the implementation. This finite state machine is an abstracted finite state machine since it doesn't include all variables, only the ones considered relevant to the current validation. If a corresponding design anomaly is itself present in the implementation, the implementations' abstracted state vector will go through an equivalent sequence to that found in the design validation done earlier. In this case the work proceeded by outputting each state vector for the executing implementation. The output list was then manually examined line by line to look for the presence of anomaly states.

The input sequence program that was incorporated into the HSS Tcl interface program to check for the presence of anomalies in the implementation is shown in Figure 6.

IP	Mnemonic
800	BEGIN
803	NOP
805	NOP
807	NOP
809	NOP
80b	NOP
80d	MARK
80f	NOP
811	NOP
813	NOP
815	NOP
817	NOP
819	MARK
81b	NOP
81d	NOP
81f	NOP
821	NOP
823	NOP
825	END

Figure 6 Sequence Validation Sequence

To keep the analysis as straight forward as possible, each instruction was executed on one-second boundaries. A HSS Tcl interface program was written to generate the output state vector sequence of the abstracted implementation state machine. Schematically, the overall process is shown in Figure 7.

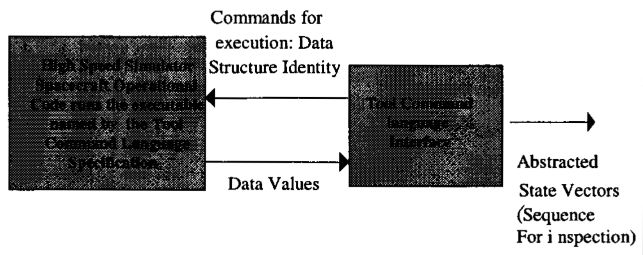
High Speed Simulator Spacecraft Operational Code runs the executable named by the Tool Command Language Specification
Commands for execution: Data Structure Identity
Data Values
Tool Command language Interface
Abstracted State Vectors (Sequence For i nspection)

Figure 7: Implementation Abstracted State Machine

The implementation was validated at this point by simply looking at the results of the simulation by hand and recognizing that a design anomaly was or was not reproduced in the output. This means visually examining the output sequence labeled "Abstracted State Vectors" to check the rollback process functionality. Two of the three anomalies found in the design validation were present in the implementation. A brief summary of the results follows.

6.3 Implementation Anomaly Validation Results

The first anomaly resulted from repetitive errors that caused the prime and the online system to get out of synchronization. Our design anomaly fault scenario required a series of prime-fault-repair sequences each of one-second duration or less. We did not see the first anomaly in the system. Further investigation with system engineers revealed that all faults take at least several minutes to repair. Therefore, repair time was extended so that anomaly one would not be seen.

The second anomaly occurs when a fault occurs less than three seconds after the sequence ends. In this case, there is no rollback. That is, once the sequence has been completed there is no rollback in response to an error injected inside the three-second-rollback window. Therefore, there is no guarantee that all instructions at the end of the sequence would have been carried out by the spacecraft. Accordingly, on this basis, the last instruction in the program should have been identified as a rollback point. Our technique demonstrated that the second anomaly was present in the implementation.

The third anomaly results from a fault that brings the prime system down when its aging buffer contains a mark point rollback address that has been aged by two seconds. According to our model checking validation, this information would not get to the online system until the following second, thereby causing its two deep online buffer to age its rollback address by an additional second. Consequently, its rollback address would be consistent with a three-second delay following a mark point when only two seconds had elapsed since the prime string had executed its last instruction.

Prime system failure was again caused by cold booting the prime string at the point it had aged its mark point by two seconds. The subsequent rollback in the new prime system did not match the old prime's rollback address. Accordingly, our technique demonstrated that the third anomaly was present in the implementation.

The cold boot process is equivalent to the injection of a single fault that brings the prime system down. This process causes the overall spacecraft controller to fail to conform to requirements since control in the new prime system rolls back to an inappropriate location. Therefore, our technique also demonstrated that the overall system made up of prime and backup systems was not single fault tolerant.

All of these results were taken with respect to the spacecraft software as it existed on the High Speed Simulator.

7 Discussion

The analysis technique used in this study is relatively new, and was not sufficiently mature just a few years ago to enable its use. The DRS operates as a communication system that must be robust under the incidence of arbitrary faults. The validation of requirements for such fault tolerant systems is particularly hard, because of the non-determinacy introduced by the fault behavior. Holzmann [10] points out that even for relatively simple communication protocols:

"It is almost impossible to manually verify correctness requirements such as the ones discussed, no matter how diligent or disciplined the designer. The behavior of even simple protocol systems can be of a complexity that no designer can be expected to assess accurately."

Worse still, the desired validation cannot be established through rigorous testing of the implementation either. The complexity of the communication system, together with the non-deterministic occurrence of faults makes exhaustive testing infeasible.

The use of model checkers opens up new possibilities for validating such systems. In principle, exhaustive checking of the requirements model is also infeasible. However, by exploiting the structure of the state space, a partial model can be extracted that is sufficient for the validation exercise. The reduction in the size of the state space was critical in this case study, and was achieved by dividing the requirements into 5 partitions and abstracting away extraneous detail. The original (reduced) estimate of the size of the model state space was over 100 million states. Although the estimate after simplification was between about 62,000 and 800,000 states, the actual number of states in the model was just over 100,000 states, allowing the validation of each of the six requirements in partition 1 to be completed in 30 seconds.

The complexity of the validation exercise was also reduced by validating requirements individually. It is possible to combine requirements (and domain properties), as described in Appendix A, so that they can be checked in a single validation run. However, doing so often increases the complexity of the model beyond the limit of current model checking technology. Hence, we only combine requirements in this way when they are known or suspected not to be independent.

It is important to note that with this approach, any claims of completeness are sacrificed; we are only performing partial validation of partial specifications. Hence, the focus is not on proving correctness, but on revealing errors [13]. We have shown in the case study that the approach is capable of finding subtle errors that are otherwise almost impossible to detect. If we did not find any errors, that would not establish correctness, but it does provide a higher level of assurance than is otherwise possible.

8 Summary and Conclusions

We have demonstrated through a case study how fault tolerance requirements can be validated through non-deterministic model checking. The system described in the case study used a mark and rollback scheme to implement fault tolerance. The system has to complete high priority tasks called critical sequences efficiently and at the same time respond to and repair faults. To meet this requirement, hard rollback points (markpoints) are embedded in the critical sequence code so that completed subtasks would not have to be repeated when fault conditions force the executing critical sequence to suspend operation to service the fault.

Faults occurring within subtasks are repaired and rollback is then done to the start of the last uncompleted subtask. A hot backup (the 'online string') is operational synchronously to increase reliability and availability.

The validation scheme described in this paper was implemented as a Spin model with three key components. First, the model contains an underlying operating system (executive) that contains a checkpointing scheme referred to as the mark and rollback process, which was modeled deterministically. Second, a generalized critical sequence was chosen to be executed by the model operating system to make it possible for requirements and design errors to surface. Finally, a fault injection process was used to non-deterministically inject a single fault into the system model. The validation system then attempted to execute the critical sequence and to recover from all possible injections of a single fault into the executing critical sequence. In this way three anomalies were discovered.

The model was reduced to a feasible size for validation by abstracting away unnecessary detail leaving behind a partial specification. The functional rollback requirement was elaborated into 6 separate but dependent requirements. A Linear Temporal Logic scheme was developed to validate three pairs of coupled requirements over the dually redundant system. This procedure allowed the rollback requirement in the prime or control system to be validated together with its coupled ancillary mirror rollback requirement in the online (hot backup) system. In this way, the study showed that a partial specification for a complex spacecraft controller can be effectively validated within the framework of the remaining requirements.

Validating all six rollback requirements in one validation run would have added a large amount of execution time to the validation. This is because the resulting LTL automaton that gets coupled to the model contributes exponentially to the size of the model. Accordingly, by partitioning the overall rollback requirements into three equivalence classes with two rollback requirements each significantly reduced the model in size (state space) and consequently in execution time as well. As a byproduct, the analysis of the results was much more straightforward since one type of rollback could be analyzed at a time.

Having completed the work of finding the three design errors using model checking, we then validated the software implementation for the presence of the three anomalies found in the design. Using a spacecraft simulator running code identical to the real spacecraft, two of the three design anomalies were found to be present in the implementation. The method is efficient since the design state space search problem is generally a relatively fast process. Whereas, the state space of a software implementation is potentially too large to search exhaustively or even reliably partially. Having found design errors by rapid search, efficient implementation validation can be done by checking for the presence of a small number of individual design anomalies

We plan to extend the application of the methodology demonstrated here to developmental efforts over the software lifecycle using partial specifications and their associated co-evolving prototype implementations. We are exploring two different approaches. The first approach works by instrumenting a partial or complete implementation in order to detect the presence of paths through the state space that correspond to the satisfaction of functional requirements. The resulting log files are then transformed into a set of traces to be executed by a model checker to validate that the implementation preserves the key properties. The functional requirements in the system are validated by expressing them as Linear Temporal Logic propositions that are translated into an appropriate automata type supported by the particular model checker in use. Then, by traversing the annotated log files encapsulated as processes over the model, the functional requirements are validated in the usual way by the model checker as discussed by Holzmann [2].

The second approach is to use the model checker to generate runtime monitors that may be embedded in the implementation. In this approach, we express correctness properties as LTL formulae, and use Spin to generate a C-encoded procedure from the formula, which is then included as a run-time monitor inside the growing implementation. The implementation is then instrumented, by hand, to inform the monitor at the occurrence of the events that the monitor is interested in, namely those events that can cause a change in the truth-value of the correctness property. The monitor would complain if it ever saw an execution that violated a stated correctness property.

This first of these approaches has been successfully used on a pilot project to validate a complex communications protocol called RMP [14]. Two teams consisting of an Independent Verification and Validation (IV&V) team and a software development team were used. Both the development team and the IV&V teams worked from an evolving partial specification. While the development team was responsible for the implementation, it was the responsibility of the IV&V team to apply a modeling scheme to check that the evolving specification and the implementation were consistent with each other. The IV&V team then used the model checker to validate the requirements. In this way when errors in the implementation surfaced they could be brought up to date with the specification; and if the specification were in error the implementation could be used to update the specification. Each derived or added requirement would, of course, then be incrementally validated and used to assist in driving the specification forward and so on. By working in tandem in this way, costly backtracking errors are prevented. The result was a saving in operational efficiency and lower maintenance costs due to good underlying design.

9 References

[1] W. Chan, R. J. Anderson, P. Beame, and David Notkin, "Improving Efficiency of Symbolic Model Checking for State-Based System Requirements", Proceedings of the ACM SIGSOFT International Symposium on Software Testing and Analysis, (ISSTA'98), Clearwater Beach, Florida, March 1998.

[2] G. J. Holzmann, "The Model Checker Spin," *IEEE Transactions on Software Engineering*, vol. 23, pp. 279-295, 1997.

[3] C. Heitmeyer and D. Mandrioli, "Formal Methods for Real-time Computing: An overview," in *Formal Methods for Real-time Computing*, C. Heitmeyer and D. Mandrioli, Eds. Chichester, UK: J. Wiley, 1996, pp. 1-32.

[4] N. G. Leveson, M. P. E. Heimdahl, H. Hildreth, and J. D. Reese, "Requirements Specification for Process Control Systems," *IEEE Transactions on Software Engineering*, vol. 20, 1984.

[5] K. L. Heninger, "Specifying Software Requirements for Complex Systems: New Techniques and Their Application," *IEEE Transactions on Software Engineering*, vol. 6, pp. 2-13, 1980.

[6] S. Easterbrook and J. Callahan, "Formal Methods for V&V of partial specifications: An experience report," *Proceedings, Third IEEE Symposium on Requirements Engineering (RE'97)*, Annapolis, Maryland, 5-8 January 1997.

[7] Z. Manna and A. Pnueli, "Tools and rules for the practicing verifier," Department of Computer Science, Stanford University, Technical Report CS-TR-90-1321, 1990.

[8] K. L. McMillan, "Symbolic model checking - an approach to the state explosion problem," in *School of Computer Science*. Pittsburgh, PA: Carnegie Mellon University, 1992.

[9] J. R. Burch, E. M. Clarke, and D. E. Long, "Symbolic Model Checking for Sequential Circuit Verification," *IEEE Transactions on Computer Aided Design of Integrated Circuits and Systems*, vol. 13, pp. 401-424, 1994.

[10] P. Ramanathan and K. G. Shin, "Use of Common Time Base for Checkpointing and Rollback Recovery in a Distributed System," *IEEE Transactions on Software Engineering*, vol. 19, pp. 571-583, 1993.

[11] G. J. Holzmann, *Design and Validation of Computer Protocols*: Prentice Hall, 1991.

[12] D. Harel, "Statecharts: A Visual Formalism for Complex Systems," *Science of Computer Programming*, vol. 8, pp. 231-74, 1987.

[13] D. Jackson and C. A. Damon, "Elements of Style: Analysing a software design with a counter-example detector," *International Symposium on Software Testing and*

Analysis (ISSTA'96), San Diego, CA, 8-10 January 1996.

[14] J. R. Callahan and T. L. Montgomery, "An Approach to Verification and Validation of a Reliable Multicasting Protocol," *International Symposium on Software Testing and Analysis (ISSTA'96)*, San Diego, CA, 8-10 January 1996, pp. 187-194.

[15] J. R. Büchi, "On a Decision method in restricted second-order arithmetic," *Proceedings of the International Conference on Logic Methodology and Philosophy of Sciences*, 1960, Stanford University Press, pp. 1-11.

[16] Reinholtz, WK and Robison, WJ., III, "The ZIPSIM series of high-performance, high fidelity spacecraft simulators," *Proceedings AIAA/Utah State University Annual Conference on Small Satellites,* Aug 29-sept 1, 1994.

[17] Patel, K and Reinholtz, W and Robison, W, "High-speed simulator: A simulator for all seasons", *Proceedings International Symposium on Space Mission Operations and Ground Data Systems (SPACEOPS96,* Munich, Germany Sept 16-20 1996; pg. 749-756

10 Appendix A: Linear Temporal Logic Background

The Spin/PROMELA modeling scheme derives much of its power from its ability to incorporate formal theorem proving elements into its search schemes. Büchi [14] discovered the fundamental relationship between finite automata and the second-order monadic calculi. This innovation made it possible to incorporate Linear Temporal Logic (LTL) assertions as components of computer modeling schemes.

A Büchi automaton is a non-deterministic Finite State Machine (FSM) A = (Σ, $\mathcal{S}$ $\mathfrak{T}$, S_0, $\mathfrak{F}$). Σ is the input alphabet, $\mathcal{S}$ is the set of states, S_0 the set of initial states, and $\mathfrak{F}$ is the set of *accepting* states. $\mathfrak{T} \in \mathcal{S} \times \Sigma \times \mathcal{S}$ is the transition relation. If (s, σ, s') $\in \mathfrak{T}$ then A can move from s to s' upon reading σ. An input word is an infinite sequence $\sigma = \sigma_1, \sigma_2, \sigma_3, \ldots, \sigma_i \in \Sigma$, while a *run*, r, over σ is an infinite sequence $s_0 \xrightarrow{\sigma_1} s_1 \xrightarrow{\sigma_2} \ldots$, where $s_0 \in S_0$, $(s_i, \sigma_{i+1}, s_{i+1}) \in \mathfrak{T}$, $i = 0, 1, \ldots$. A run, r, is said to be accepting iff there exists a state $g \in \mathfrak{F}$ such that g appears infinitely often in r. The *language* $\mathcal{L}$(A) is the set of all input words, σ, such that A has an accepting run over σ.

Let f_i be an LTL assertion corresponding to a system requirement to be validated that generates automaton A_i. Given n Büchi automata of the form $A_i = (\Sigma_i, \mathcal{S}_i, \mathfrak{T}_i, S_{0i}, \mathfrak{F}_i)$, they are closed under the operation of intersection. Their intersection $\bigcap_{i=1}^{n} A_i$ accordingly is a Büchi automaton, and it accepts the language $\bigcap_{i=1}^{n} L(A_i)$. The LTL formula that generates this automaton has the form

$$f = \bigwedge_{i=1}^{i=n} f_i \tag{1}$$

Equation (1) allows multiple LTL formulae to be concatenated such that the resulting automaton will preserve the characteristics of the language accepted by each automaton were it to be implemented in isolation. This means that the set of all input words, σ, that were recognized by each automaton A_i in isolation will also be recognized by the composite automaton $\bigcap_{i=1}^{n} L(A_i)$.

By incorporating the Finite State Machine (FSM) representation of the formal properties to be validated by the model, the model can be routinely checked for the presence or the absence of the desired characteristics.

The Spin/PROMELA system has an LTL translator that can produce the corresponding Büchi automaton from an input requirement expressed as an LTL formula. The Spin modeling system checks to see that finite state program $\mathcal{P}$ satisfies the temporal logic formula f. First, the global state graph of $\mathcal{P}$ is computed. Second, the Büchi automaton is constructed for $\neg f$: $A_{\neg f}$. Third, the synchronous product $\mathcal{P} \times A_{\neg f}$ is computed. Finally, the validation run is performed on $\mathcal{P}$

$\times$ $A_{\neg f}$. For each state transition in $\mathcal{P}$, Spin checks to see if a corresponding transition in $A_{\neg f}$ is possible. Once one of the accepting states of $A_{\neg f}$ has been entered, it must be shown that that state is reachable from itself. When this happens, $A_{\neg f}$ will have been shown to have recognized a string σ from the language generated from the original LTL formula $\neg f$. For efficiency, Spin executes the 3 steps in 1 pass. At this point a trail file can be written showing the sequence of state transitions in $\mathcal{P}$ that gave rise to the accepting state in $A_{\neg f}$. This file can then be annotated and run as a test case against the implementation.